Analisis y Diseño de Algoritmos

Juan Miguel Rojas

14 de mayo de 2024

1. Ejercicio 2-1: Generalizations of SubsetSum (Erickson, página 93).

Describe recursive algorithms for the following generalizations of the SubsetSum problem:

(a) Given an array $X[1..n]$ of positive integers and an integer T, compute the number of subsets of X whose elements sum to T.

Solución

Especificación del problema

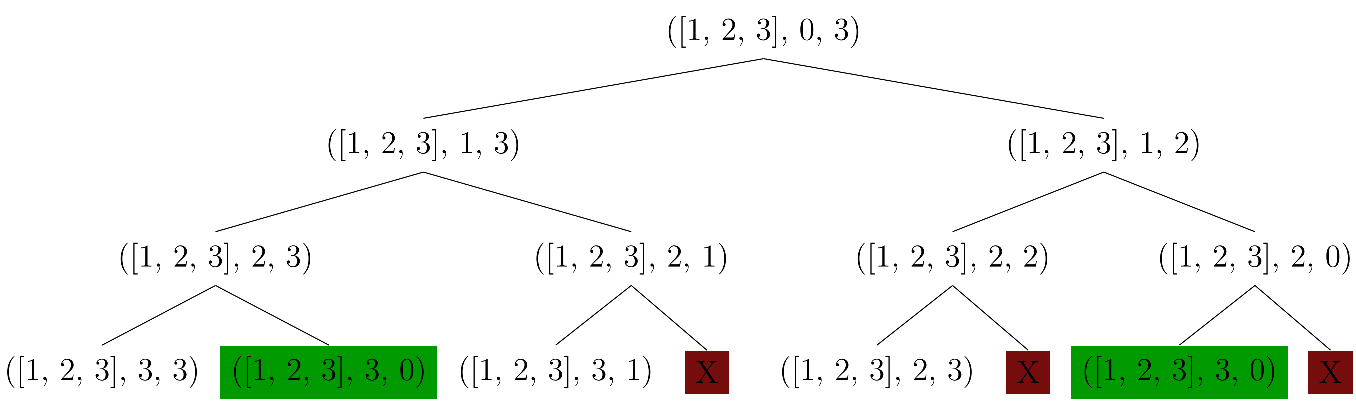
Entrada: Una lista $X[1...n]$ de numeros enteros positivos y un entero T
Salida: ¿Cuales y cuantos subconjuntos de X su suma es T?

A través de backtracking podemos abordar este problema. Pero existen 2 caminos posibles para llegar a la solución. Y es que realmente podemos considerar podar o no podar el arbol y en mi solución presenta las siguientes diferencias.

- En el caso en el que utilizo poda tengo 240957929287844081848684965351423170282681927338 posibles soluciones (aproximadamente), es decir, cuando recurri contabilice la cantidad de resultados que puedo evaluar.
- En el caso en el que no utilizo la poda tengo practicamente 3 veces el valor anterior de posibles soluciones.

Por lo cual, con este analisis me doy cuenta de lo importante que es tener una buena poda a la hora de hacer backtracking o reintento, ya que esto me permite reducir el costo que me genera esta tecnica.
A continuación tenemos un ejemplo en backtracking para un conjunto $X = [1, 2, 3]$ y $T = 3$

Arbol con poda



Función objetivo

Sea $0 \leq i \leq N$

$\phi(i, data, ans)$: Todas los posibles subconjuntos de X que su suma sea como resultado T.

Reformulación de la especificación

Entrada: Una lista $X[1...n]$ de numeros positivos y un entero T
Salida: $\phi(0, list(), 0)$.

Codigo

Listing 1: SubsetSum con poda

```
def solve(X, T, i, data, ans):
  if(i == len(X) and T == 0):
    print(data, T == 0)
    ans += 1
  elif(i == len(X) and T != 0):
    ans = 0
  else:
    ans = solve(X, T, i + 1, data, ans)
    if(X[i] <= T):
      data.append(X[i])
      ans += solve(X, T - X[i], i + 1, data, ans)
      data.pop()

  return ans
```

Listing 2: SubsetSum sin poda

```
def solve2(X, T, i, data, ans):
  if(i == len(X) and T == 0):
    print(data, T == 0)
    ans += 1
  elif(i == len(X) and T != 0):
    ans = 0
  else:
    data.append(X[i])
    ans = solve2(X, T - X[i], i + 1, data, ans)
    data.pop()
    ans += solve2(X, T, i + 1, data, ans)
```

```
  return ans
```

Complejidad

Temporal: La complejidad del algoritmo es exponencial $O(2^n)$ ya que puede probar todos los posibles subconjuntos posibles en el peor de los casos. Por lo tanto, la complejidad es exponencial.
Espacial: $O(1)$

Pruebas

Para esta sección, realice pruebas y podemos observar los resultados de la siguiente manera

Casos de prueba

```
def main():
  X = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
  T = 20
  print(solve(X, T, 0, [], 0)) #Solucion con poda
  print(solve2(X, T, 0, [], 0)) #Solucion sin poda

main()
```

Salida

```
Solucion con poda:
[5, 7, 8]
[5, 6, 9]
[4, 7, 9]
[4, 6, 10]
[3, 8, 9]
[3, 7, 10]
[3, 4, 6, 7]
[3, 4, 5, 8]
[2, 8, 10]
[2, 5, 6, 7]
[2, 4, 6, 8]
[2, 4, 5, 9]
[2, 3, 7, 8]
[2, 3, 6, 9]
[2, 3, 5, 10]
[2, 3, 4, 5, 6]
[1, 9, 10]
[1, 5, 6, 8]
[1, 4, 7, 8]
[1, 4, 6, 9]
[1, 4, 5, 10]
[1, 3, 7, 9]
[1, 3, 6, 10]
[1, 3, 4, 5, 7]
[1, 2, 8, 9]
[1, 2, 7, 10]
[1, 2, 4, 6, 7]
[1, 2, 4, 5, 8]
[1, 2, 3, 6, 8]
[1, 2, 3, 5, 9]
[1, 2, 3, 4, 10]
Conteo: 31
Solucion sin poda:
```

```
[1, 2, 3, 4, 10]
[1, 2, 3, 5, 9]
[1, 2, 3, 6, 8]
[1, 2, 4, 5, 8]
[1, 2, 4, 6, 7]
[1, 2, 7, 10]
[1, 2, 8, 9]
[1, 3, 4, 5, 7]
[1, 3, 6, 10]
[1, 3, 7, 9]
[1, 4, 5, 10]
[1, 4, 6, 9]
[1, 4, 7, 8]
[1, 5, 6, 8]
[1, 9, 10]
[2, 3, 4, 5, 6]
[2, 3, 5, 10]
[2, 3, 6, 9]
[2, 3, 7, 8]
[2, 4, 5, 9]
[2, 4, 6, 8]
[2, 5, 6, 7]
[2, 8, 10]
[3, 4, 5, 8]
[3, 4, 6, 7]
[3, 7, 10]
[3, 8, 9]
[4, 6, 10]
[4, 7, 9]
[5, 6, 9]
[5, 7, 8]
Conteo: 31
```

(b) Given two arrays $X[1..n]$ and $W[1..n]$ of positive integers and an integer T, where each $W[i]$ denotes the weight of the corresponding element $X[i]$, compute the maximum weight subset of X whose elements sum to T. If no subset of X sums to T, your algorithm should return -∞.

Solución

Especificación del problema

Entrada: Dos listas $X[1..n]$ y $W[1..n]$ de numeros enteros positivos y un entero T, donde $W[i]$ denota los pesos de $X[i]$
Salida: El subconjunto de peso maximo en X y su suma es T

Para solucionar este problema, opté por manejar dos variables globales las cuales me permetiran tener el control del subconjunto de peso maximo que su suma sea T. Como este problema es un problema de optimización debemos hallar un unico valor.
En este caso, también se logro resolver el problema con poda y sin poda. Como mencione en el anterior ejercicio, la cantidad de ramas que se producen al resolver el problema sin poda son bastantes dependiendo de la entrada, y la poda sigue siendo un factor importante.

Función objetivo

Sea $0 \leq i \leq N$
$\phi(i, MW, data, ans)$: El subconjunto de X que su suma sea T, de peso maximo de X.

Reformulación de la especificación

Entrada: Dos listas $X[1..n]$ y $W[1..n]$ de numeros enteros positivos y un entero T, donde $W[i]$ denota los pesos de $X[i]$
Salida: $\phi(0, 0, list(), 0)$.

Codigo

Listing 3: SubsetSum con pesos y con poda

```python
def solve3(X, W, T, MW, i, data, ans):
  global value
  global subconjunto
  if(i == len(X) and T == 0):
    if(MW > value):
      subconjunto = data.copy()
      #print(subconjunto)
    value = max(value, MW)
    #print(data, T == 0)
    ans += 1
  elif(i == len(X) and T != 0):
    ans = 0
  else:
    ans = solve3(X, W, T, MW, i + 1, data, ans)
    if(X[i] <= T):
      data.append((X[i], W[i]))
      ans += solve3(X, W, T - X[i], MW + W[i], i + 1, data, ans)
      data.pop()
```

```
  return ans
```

Listing 4: SubsetSum con pesos y sin poda

```
def solve4(X, W, MW, T, i, data, ans):
  global value
  global subconjunto
  if(i == len(X) and T == 0):
    if(MW > value):
      subconjunto = data.copy()
      #print(subconjunto)
    value = max(value, MW)
    #print(data, T == 0)
    ans += 1
  elif(i == len(X) and T != 0):
    ans = 0
  else:
    #print(i, len(X), MW, T,  T != 0)
    data.append((X[i], W[i]))
    ans = solve4(X, W, MW + W[i], T - X[i], i + 1, data, ans)
    data.pop()
    ans += solve4(X, W, MW, T, i + 1, data, ans)

  return ans
```

Complejidad

Temporal: La complejidad del algoritmo es exponencial $O(2^n)$ ya que puede probar todos los posibles subconjuntos posibles en el peor de los casos. Por lo tanto, la complejidad es exponencial.
Espacial: $O(1)$

Pruebas

Para esta sección, realice pruebas y podemos observar los resultados de la siguiente manera

Casos de prueba

```
def main():
  X = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
  T = 20
  W = [1, 1, 1, 1, 1, 1, 1, 1, 50, 50]
  global value
  global subconjunto
  value = -float('inf')
  if(solve3(X, W, T, 0, 0, [], 0) != 0):
    print(value)
    print(subconjunto)
  else:
    print(value)

  if(solve4(X, W, 0, T, 0, [], 0) != 0):
    print(value)
    print(subconjunto)
  else:
    print(value)

main()
```

Salida

```
Solucion con poda:
Suma de pesos: 101
[(1, 1), (9, 50), (10, 50)]
Solucion sin poda:
Suma de pesos: 101
[(1, 1), (9, 50), (10, 50)]
```